CSC2/455 Software Analysis and Improvement
Introduction to Loop Optimizations

Sreepathi Pai

URCS

March 18, 2019

Outline

Outline

Optimizations

- Part I: Analysis
  - Iterative Dataflow Analysis
  - SSA Form
- Part II-A: Optimization
  - Dead Code Elimination
  - Partial Redundancy Elimination
- Part II-B: Loop Optimizations
  - Dependence Analysis
  - Loop Transformations
- Part III: Code Generation
  - Instruction Selection
  - Instruction Scheduling
  - Register Allocation
- Part IV: Advanced Topics

Remainder of the course

- Loop Optimization
- Use LLVM
- Advanced Topics
  - Interprocedural Analysis
  - Type Inference
  - Abstract Interpretation
  - Program Verification
  - more, depending on time ...
- CSC455 paper reading
  - 25% of final exam grade based on paper reading

Outline

Why Loop Transformations

- Potentially lots of computation
  - A few operations execute many times
- Potentially lots of memory accesses
- Array-based data structures show up frequently
  - Matrices, vectors, etc.
- Loops are naturally paired with arrays
- FORTRAN
  - FORMula TRANslator
  - World's first high-level programming language

Important Applications

- Scientific Computing/Computational Science
  - Simulation of Galaxies, Molecules, etc.
  - Drug Discovery
- Audio/Video Processing
  - Signal Processing
  - Compression
- Machine Learning (specifically Deep Learning)
  - Recognizing cats
  - Showing targeted ads

Matrix Multiply – IJK

- Multiplying two matrices:
  - A $(m \times n)$
  - B $(n \times k)$
  - C $(m \times k)$ [result]
- Here: $m = n = k$

```
for(ii = 0; ii < m; ii++)
   for(jj = 0; jj < n; jj++)
      for(kk = 0; kk < k; kk++)
         C[ii * k + kk] += A[ii * n + jj] * B[jj * k + kk];
```

Matrix Multiply – IKJ

```
for(ii = 0; ii < m; ii++)
   for(kk = 0; kk < k; kk++)
      for(jj = 0; jj < n; jj++)
         C[ii * k + kk] += A[ii * n + jj] * B[jj * k + kk];
```

Performance of the two versions?

- on 1024x1024 matrices of `ints`
- which is faster?
- by how much?

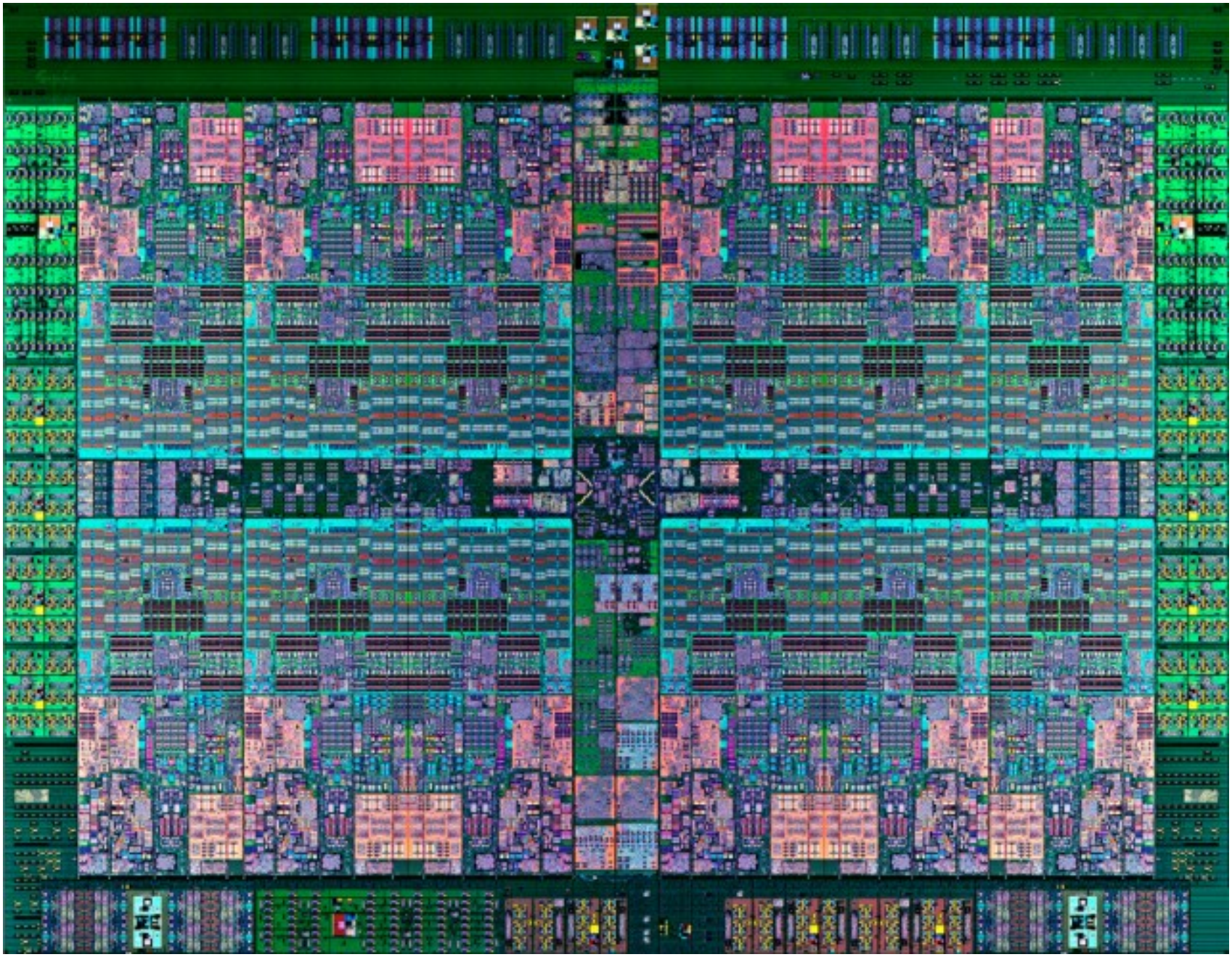
Performance of the two versions

- on 1024x1024 matrices
- Time for IJK: 0.554 s $\pm$ 0.003s (95% CI)
- Time for IKJ: 6.618 s $\pm$ 0.032s (95% CI)

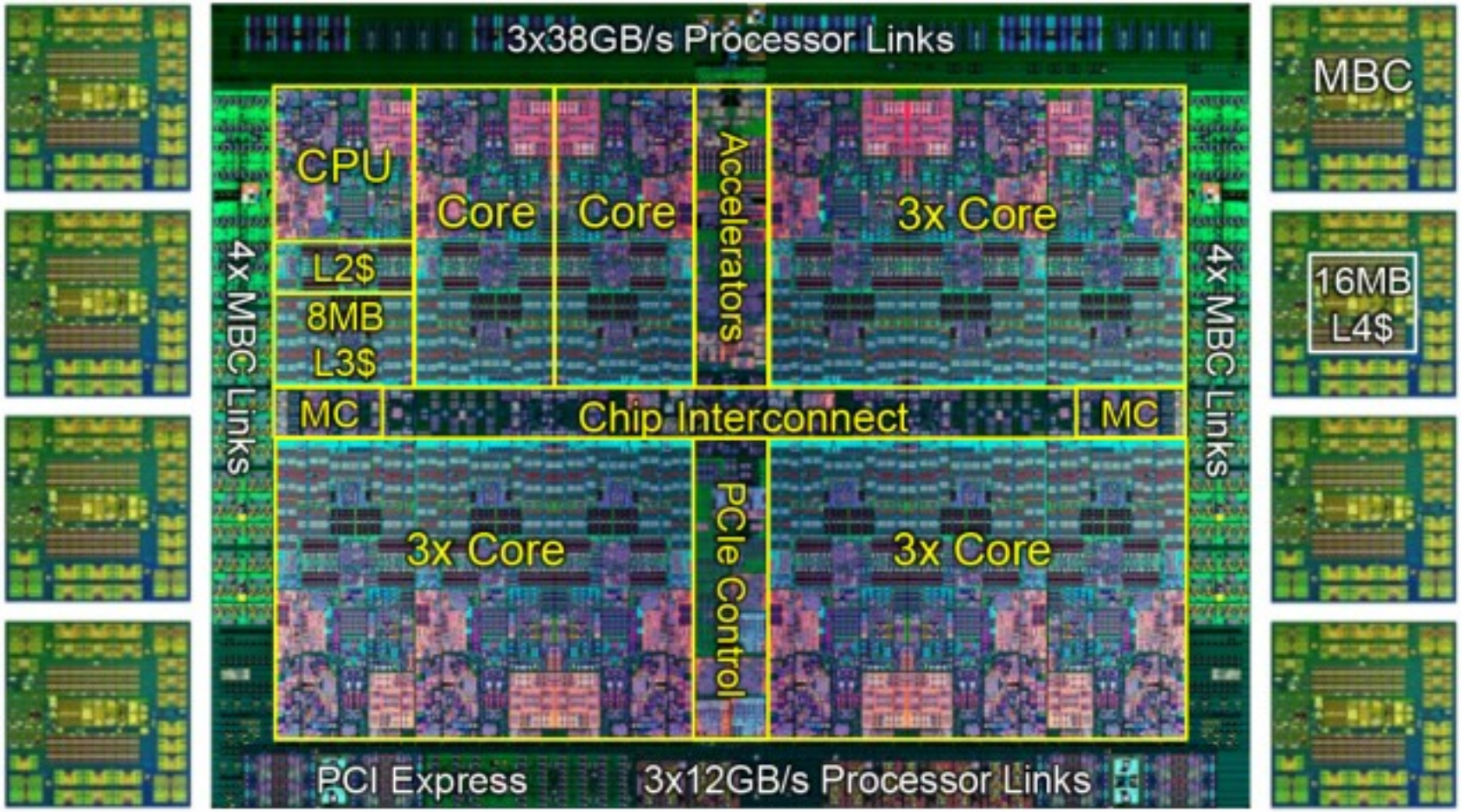
What caused the nearly 12X slowdown?

- Matrix Multiply has a large number of arithmetic operations
  - But the number of operations did not change
- Matrix Multiply also refers to a large number of array elements
  - Order in which they access elements changed
  - But why should this matter?

Die shot of a processor (IBM Power 8)



Die shot of a processor (IBM Power 8)



extremetech

Motivation for a memory hierarchy

- Not all memory types are equal
  - Consider: SRAM, DRAM and magnetic storage
- Speed to access data
  - Depends on size and type of memory
  - SRAM $>$ DRAM $>$ Magnetic storage
- Density of storing data
  - Bits per square millimeter
  - SRAM $<$ DRAM $<$ Magnetic storage

The Memory Hierarchy – Part I

- Registers
  - managed by compiler
  - "logic"
- L1 cache
  - small (10s KB), usually 1-cycle access
  - SRAM (also "logic")
- L2 cache
  - largish (100s KB), 10s of cycles
  - SRAM
- ...

The Memory Hierarchy – Part II

- L3 cache
  - usually on multicores
  - much larger (MB), 100s of cycles
  - SRAM or (recently) embedded DRAM
- DRAM
  - off-chip, large (GB)
- HDD
  - Magnetic/Rotating Storage (TBs)
  - Flash memory (GBs)

Performance of the hierarchy?

Why structure memory in a hierarchy?

- Each level of hierarchy adds a delay
- Time to access memory increases!
  - Or does it?

Performance of the hierarchy

- Structures in memory hierarchy duplicate data stored further away
  - original meaning of the word *cache*
- If data is found closer to processor (i.e. *hit*), read it from there
- Otherwise (i.e. *miss*), pass request to next level of the hierarchy

Why the hierarchy works in practice

- Data Reuse (or "locality")
  - Temporal (same data will be referred again)
  - Spatial (data close to each other in *space* will be referred close to each other in *time*)
- Speed differences
  - Time to access L1: 1ns
  - *Branch mispredict: 3ns*
  - Time to access L2: 4ns
  - Main memory access time: 100ns
  - SSD access time: 16μs
  - Rotating media access time: < 5 ms
  - From Latency Numbers Every Programmer Should Know

The cache equation (informal)

Assume a one-level cache (i.e. cache + RAM):

$$latency = latency_{hit}$$

or

$$latency = latency_{miss}$$

The cache equation for one level of caches

$$latency_{avg} = (fraction_{hit}) * latency_{hit} + (1 - fraction_{hit}) * latency_{miss}$$

Goal 1 of Loop Transformation: Improve Locality

Can we analyze a program's locality? Can we change the program to get better locality [and hence, better performance]?

Parallel Processing

- Our matrix was 1024x1024
  - 1 million output elements
- Each output matrix entry can be calculated independently of others
  - (Informally) Does not need other output values

Embarrassingly Parallel

- On a shared-memory machine with N processors
  - Shared memory: Each processor can "see" the same memory
  - I.e. your mobile phone and most modern desktops
- Each processor can be given $(1024 \times 1024)/N$ output elements
  - "Embarrassingly Parallel"
- Potentially reduce time by (up to) N

Embarrassingly Serial?

Consider a single processor's work:

```
for(kk = 0; kk < k; kk++)
   C[ii * k + kk] += A[ii * n + jj] * B[jj * k + kk];
```

Must this be executed serially?

Reductions

- Addition is associative
- Split up arrays into K parts
- Compute the sum of each part separately (in parallel)
- Combine the sums
  - Tree reduction

Goal 2 of Loop Transformations: Exploit Parallelism

- Known as "vectorization"
- Coarse-grain
  - Thread-level parallelism (across cores)
- Fine-grain
  - SIMD-style parallelism (within a core)

Loop Interchange

```
for(ii = 0; ii < m; ii++)
   for(jj = 0; jj < n; jj++)
      for(kk = 0; kk < k; kk++)
         C[ii * k + kk] += A[ii * n + jj] * B[jj * k + kk];
```

- 3 loops, 6 possible orderings
- All 6 orderings are "correct"
  - How do we know?
  - How can a compiler figure this out?
- The 6 orderings do not perform the same
  - How can a compiler analyse this?

When are Loop Transformations Correct?

- Loosely speaking, loop transformations change ordering of operations in loops
  - to improve locality
  - to increase parallelism
- These transformations are legal only if:
  - (too restrictive) they preserve the semantics of the original program
  - (less restrictive) they preserve the *dependences* of the original program

Outline

Next class

- Dependence Analysis
- Computational Geometry

References

- Dragon Book, Chapter 11
- Allen and Kennedy